Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Štefan Gurský

C library for symbolic manipulation

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.
Kabinet software a výuky informatiky

Studijní program: Informatika, Správa počítačových systémů

2008

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I hereby declare that this text is my own work. Where other sources of information have been used, they have been acknowledged. I agree with making this work publicly available.

Rád by som poďakoval vedúcim práce, Tomášovi Holanovi a Ondřejovi Čertíkovi za to, že táto práca bola včas vypísaná a za trpezlivosť a netrvaní na termínoch.

I would also like to express my thanks to my supervisors Tomáš Holan and Ondřej Čertík for setting this bachelor thesis just in time and for their patience and not insisting on deadlines.

V Praze dne

Prague Štefan Gurský

Contents

Title: C library for symbolic manipulation

Author: Štefan Gurský

Department: Department of Software and Computer Science Education

Supervisor: RNDr Tomáš Holan Ph.D.

Supervisor's email address: holan@ksvi.mff.cuni.cz

Consultant: Bc. Ondřej Čertík

Consultant's email address: ondrej@certik.cz

Abstract: There are many programs doing symbolic manipulation with mathematical expressions. They are called Computer Algebra Systems and most of them contain some programming language. In this bachelor thesis a library for C programming language was created that allows using C as a language for symbolic manipulation. With provided functions user can store mathematical expressions in C variables and work with them in an easy and convenient way. It is possible to build expressions containing standard mathematical operations and functions, reduce them, expand them, find derivatives and Taylor polynomials. The library (called SymCe) is easy to understand not only for user, but also for a programmer that wishes to see its inner working and possibly extend it. Simplicity is the main feature that distinguishes SymCe from other similar pieces of software.

keywords: symbolic manipulation, CAS, mathematics, library

The aim of this bachelor thesis is to write a simple library for symbolic manipulation. This library should be very easy to use and understand; it should follow the KISS (keep it simple stupid) principle. Simplicity is the main aspect of the library. The name was chose to be SymCe.

1 Symbolic manipulation and CAS

Symbolic manipulation in this text means working with mathematical expressions, transforming them to equivalent forms and creating new expressions either by combining those that already exist using operators and functions or by other transformations. The software that does this is usually called CAS (meaning *Computer Algebra System*).

There are several computer programs that can do symbolic manipulation. Some of them are free (such as *Yacas*, *Axiom* or *Maxima*), some of them are not (the most famous and probably the most expensive one being *Wolfram Mathematica*). All of these programs have several common features. Their interface presents user with a large window where user types commands and gets answers from the computer. These commands may vary from very simple (such as `1+1`) to fairly complex.

All of computer algebra systems provide some kind of programming language for users to program advanced calculations and to avoid entering repeating commands all the time by hand.

However, there are some problems with these programming languages. Programming language of a computer algebra system is specific to that particular CAS and cannot be used in any other CAS. If user A wants to share a program with user B they need to have the same CAS system. If user B does not have it, it can be quite expensive to get it or to rewrite the original program to the programming language of B's system.

It can even sometimes happen, that program written for particular version of CAS will not run as expected in another version of the same system, because the author of the CAS decided to change properties of some function or because the function was removed completely.

Many examples of how the CAS programming languages are not on a par with real programming languages can be found in "*Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language*". These examples include problems with variable scope ("*Even explicitly declaring a variable to be local does not always spare the programmer surprises*") or unavailability of structured data.

That is why people started to look at real programming languages and think about using them for doing symbolic computations.

1.1 CAS in real programming languages

Real here means that these programming languages were designed to create computer programs without specific intentions about the kind of programs that can be written in them. Real programming languages have several advantages when compared to CAS programming languages. First and most important advantage is that they are standardized and that many people use them so they usually do not have problems of CAS programming languages, because these problems were already discovered and fixed. Real programming languages are not ambiguous[1], variables have clearly defined scopes and they tend not to change very often.

Another advantage of real programming languages is that it is much easier to find a programmer for real programming language than for a CAS programming language.

Of course, programming languages have also quite a few disadvantages when it comes to doing symbolic manipulation. They were not designed to do such work. They do not have simple ways to store mathematical expressions or mathematical objects. This is where specialized libraries come in.

1.2 Libraries for symbolic manipulation

In a CAS languages it is possible to assign mathematical expressions to variables, add, subtract, multiply, divide (and more) them using standard operators (and the result of these operations is again an expression), substitute another expression for a variable and much more. In real programming language these variables need to be of a type that is specified by a programmer. Suppose a programmer creates a variable of his own type "*Expression*" and wants to use it. However, to write something like

```
Expression a = x+2x
```

the programmer has to define operators on his or her type. And even then he or she cannot write `2x`, because in real programming languages `2x` might be either an identifier or (more often) an illegal expression (author is unaware of any real programming language where such construction would mean multiplication). However, these are only minor drawbacks and there exist libraries for symbolic manipulation in various programming languages.

Most of these libraries are written for object oriented languages, since they are more intuitive to work with (operator overloading being the most significant feature for this – it is usually more convenient to write `a+b` than `plus(a, b)`).

One example of such library is *GiNaC* (the name is a recursive acronym GiNaC is Not a CAS). It is written in C++. User uses mainly two basic types *symbol* and *ex*. *Symbol*s are mostly variables in mathematical sense and *ex*es are expressions that consist of operators, symbols and other objects. Operators are most often written as it is common in mathematics with some exceptions, most notably power operator that is often labeled `^` but overloading it would bring some undesired effects:

1 If they were, it is most probably already solved.

- Due to the binding of the operator `^`, `x^a^b` would result in `(x^a)^b`. This would be confusing since most (though not all) other CAS interpret this as `x^(a^b)`.
- Also, expressions involving integer exponents are very frequently used, which makes it even more dangerous to overload `^` since it is then hard to distinguish between the semantics as exponentiation and the one for exclusive or. (It would be embarrassing to return `1` when one has requested `2^3`.)

  Quoted from GiNaC tutorial, *Sums, products and powers*, http://www.ginac.de/tutorial/Fundamental-containers.html

Another example is *SymPy* that is written in Python. Again it has symbols and operators and functions and more. SymPy is not only library but can be used inside python command interpreter as an ordinary CAS application. All that is needed is to import it to the interpreter. Choosing Python as a language for SymPy provided some advantages, for example automatic garbage collection or large numbers (with built-in power operator).

The closest thing to SymCe seems to be *PARI/GP*, that really can use C to do the calculations; however, it uses its own language (gp) that is then transformed to C using utility called *gp2c*, so the user does not actually write the code in C (although it is possible, but it is not the primary way of using it).

2 SymCe

SymCe is an attempt to write library for symbolic manipulation in a non-object oriented programming language. Language C was chosen, because it is simple (does not have too many advanced features), widely used (there are many people using C) and applications written in it are generally highly portable.

There is of course a question if such thing as library for symbolic manipulation can even be done in a language such as C. SymCe is a proof of concept that it can be accomplished without having to go to too low level. However, there were quite a few things that needed to be sorted out.

First of the problems is “What types should be used and especially what functions will be used with them?” Of course the solution is to use own types, but should there be different types for different kinds of expressions? For example should program variables that contain mathematical variables be of a different type than variables containing more complicated expressions like this: `2*cos(x+y)`? And of course there are numbers. What kind of numbers should be used? Standard built in types such as *int* or *double*? Or should there be numbers specially designed for SymCe? Or should it use some proven implementation of numbers? The problem connected with this one is also how to write functions that work with expressions. Let’s demonstrate this problem on a simple *plus* function. If we want to, for example, create a new expression by adding two different ones, we would probably use a function such as this:

```
a = plus(b, c);
```

where `b` and `c` would both be of an expression type. But if we wanted to create `x+2`, it would not be possible to use same *plus* function but rather use some function such as *plus_expression_integer* and it would be necessary to create a function like this for every operator and for every combination of types. This is not really viable and convenient, so another approach was used.

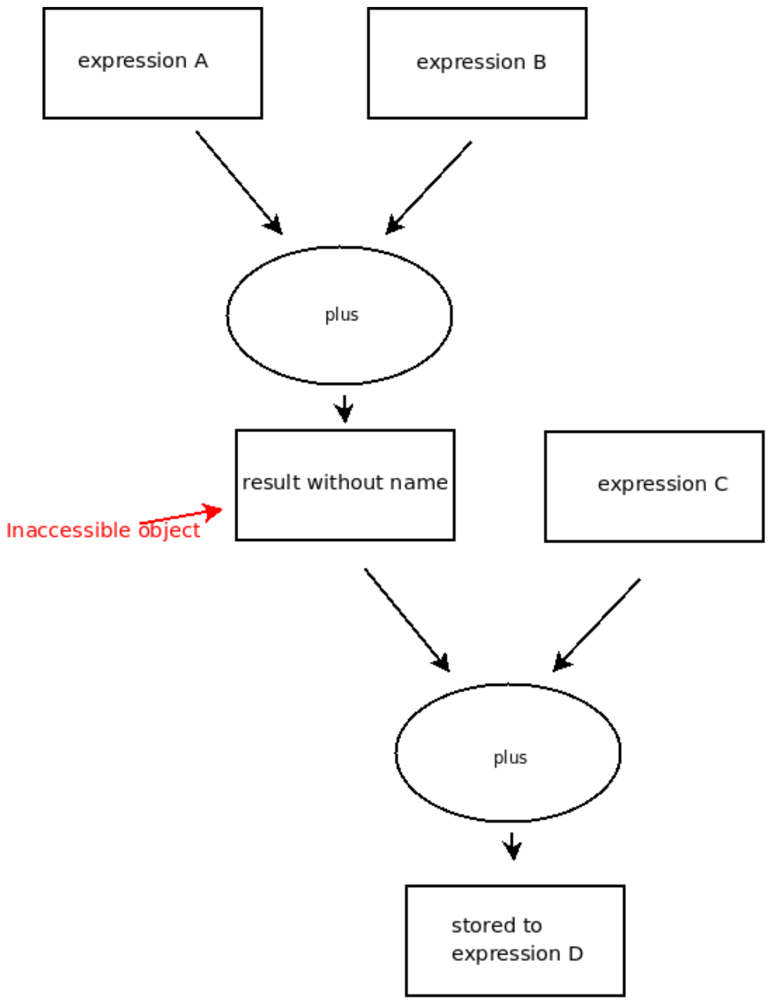
Only “expression” variables are used in operations and all other types need to be converted to them. This does not mean that you need to explicitly declare variable for every number you want to use, but you should be able to use conversion in place like this:

```
a = plus(b, convertint(2))
```

In those programming languages that have function overloading it would not be problem to have a function with one name for any types, but even in those languages it would be necessary to explicitly write all of these functions. Also with *plus* function (and some similar others) one might want to have function that sums any number of arguments of any type. While it is possible to create such a function in C, the author decided not to do it, since these functions need to get the number of parameters and their types from somewhere (meaning that user would have to provide them) and because it is not really necessary to have such function. To create expression such as `a+b+c`, user would write `plus(a,plus(b,c))` (or `plus(plus(a,b),c)`).

Both of these pieces of code should transform automatically to same expression: `a+b+c` (of course only for associative operators).

This leads to another question that has to be answered. How should the functions for building an expression return their results. Again take *plus* function as a perfect example. Suppose we want to add `a` and `b` and place result to `c`. One way to do it would be creating *plus* function as `plus(c,a,b)` with one of its parameter as an output parameter. However, it slows down creating more complex expressions, because there is always need for some temporary variable: two steps and one temporary variable – `plus(temp, a,b)` and `plus(d, temp, c)` – are needed for `d=a+b+c`. This approach has also problem with situations such as `plus(a,a,a)` and with the fact, that output parameters in C are done via pointers. While both problems are only minor, to make everything more simple, it was decided to use another approach: `c=a+b` would be written like `c = plus(a, b)`. To create `a+b+c`, one just writes `plus(a, plus(b, c))`. It is simple to understand for user and quite natural. However, it has also at least one quite serious drawback. When function such as *plus* is used, it takes arguments and creates a new expression based on them in some newly allocated memory. However, if this expression is used as a parameter in another function, it becomes inaccessible in the memory (see the following picture).



For simple objects (such as built in types or plain structures that do not contain pointers) it is not a problem, because they exist on stack and are released when function returns. However, an expression is not a simple object. It can contain pointers to other parts of the memory (to heap) and when it is deleted, these pointers would be gone and parts of memory to which they “pointed” would become inaccessible. This is called memory leak and it is a bad thing to have it in any program. If SymCe were written in C++, these problems could be solved by using destructor in expression that would be called when “result without name” shown in the picture is deleted. But in C it started to look as unsolvable problem. It is of course possible to write a function that releases all memory that is needed for an expression. But the problem is when to call it. In a situation from the previous picture, we need to delete our “result without name”. It cannot be deleted before second *plus* is run, since it is needed there. So it needs to be deleted inside the function or after it.

It is simple to release memory used by “result without name” inside the second function. It is as simple as calling some *release_expression* function on the first argument of *plus*. The problem is that *plus function does not know if its arguments are not needed after returning*. If user calls `plus(a,b)`, he or she has every right to assume that it is safe to use `a` or `b` later in the program, so that *plus* function cannot delete its parameters.

Another possibility would be using the memory of the parameters directly. In case of `plus(a, b)` the result would be expression containing `plus` operator and its operands would be pointers to the same memory where `a` and `b` point. If one of the parameters were result of some previous function, its memory would now be accessible within the newly created expression. However, this approach does not work for functions that get an expression and create completely new one as a result. They would not have a place to store its parameter within the result (and they cannot release its memory). This is also not a solution.

However, a solution appeared. Why not use something that is available in some other programming languages – a garbage collector.

2.1 C garbage collector

Garbage collector is a tool that simplifies memory management for a programmer.

When program ask system for some more memory and there is enough memory available, the program is given new memory to use it for whatever purposes. In C programming language the function that asks for memory is called *malloc*. But memory that a program can get is limited so it is necessary that program either reuses the memory it has or returns it back to the operating system when it is no longer needed. Returning memory to the operating system can be done automatically or manually using the appropriate function. In C language this is done manually and the function to release memory is called *free*. Programmer has to keep track of where his program needs memory and release it when it is no longer needed. However, sometimes programmer forgets to do it properly so program asks for new memory and old memory that is no longer needed stays allocated to the program. And eventually memory can become inaccessible, since all pointers that once did point to that memory now "have another direction". This situation is called memory leak and is really undesirable in any program.

Another possibility to manage memory and to ensure that memory that is no longer used and accessible is returned back to system, is using a garbage collector. Garbage collector keeps track of memory that program gets and when it detects that some memory can no longer be accessed, it returns that memory to the operating system. It does not always return memory immediately after it is not needed so there might be some inaccessible memory taken by the program, but it keeps amount of such memory at an acceptable level.

In several programming languages the garbage collector is part of that language and programmer can use all the advantages it provides. In C programming language the garbage collector is not a part of the language and therefore it has to be somehow "put there". Thankfully there is already easy-to-use implementation of garbage collector for C that can be found at http://www.hpl.hp.com/personal/Hans_Boehm/gc/ and which can be simply used in SymCe or in any other C program that needs it.

There, most probably, would be some way to avoid the need for garbage collector in SymCe, but using any way to avoid it would mean that user of SymCe would have to take much more care of how he or she uses the SymCe functions. Let's have a look again at "plus example". Constructions such as `plus(a, plus(b, c))` would not be possible without memory leaks. So at any time, user could call plus only once. Another type of calls that would not be possible without memory leaks would be `a = plus(a, a)`. The previous content of `a` would disappear. Other option would be placing result to output parameter. For example calling `plus(a, b)` would place result to b or adding one additional parameter. In both cases however there would be "special cases" such as `plus(a, a, a)` and it would be also inconvenient for user. Therefore garbage collector was selected as a good thing to use.

2.2 Using SymCe

To use SymCe you need to include the file `Multitree.h` in your code. This file contains the main SymCe structure called *Multitree* and prototypes of the functions that work with this structure. The *Multitree* structure can hold any SymCe expression for you. Your program variables will be pointers to this structure. For user there is no difference between pointer to *Multitree* (*Multitree* pointer) and mathematical expression.

Let's look at very simple example code that calculates first derivative of `sin(x)` with respect to `x`:

```
Multitree * x = treevariable("x");
puts(treetostring(treediff(treesin(x), x)));
```

This piece of code first creates program variable `x` that will hold mathematical variable “`x`”. It is good idea to name your program variables that hold SymCe variable with same identifier. The second line does several things. The *treesin* function creates new expression with a sine of its parameter, *treediff* finds its derivative (with respect to variable stored in the second parameter – `x` in this particular case), *treetostring* converts the whole differentiated `sin x` to C string (*char* pointer) and using standard C function *puts* the result is outputted to the screen. Program variable `x` still holds its original value (that is mathematical variable `x`). All the intermediate structures (`sin(x)`, its derivative and its representation as a string) are no longer accessible and the memory they occupied is now taken care of by garbage collector.

2.2.1 Multitree

All expressions in SymCe are stored in a *Multitree*. It can store numbers, variables, constants and expressions that are built from these using “operators” and functions. To start using a *Multitree* you first create a pointer to it:

```
Multitree * a = NULL;
```

Using SymCe functions you will later point it to expressions. As for why it is now `NULL`, it is good practice to set pointers to `NULL` if they are not yet needed. Also SymCe functions will check if the multitree pointer supplied to them is not `NULL` and will not break anything if it is (if you supply random pointer to function that expects pointer to data, there are practically no means to check whether the pointer is correctly initialized and actually points to some meaningful data).

To assign a mathematical variable to the *Multitree*, use *treevariable* function:

```
a = treevariable("a");
```

It takes one parameter of *char* * type with the name of the new variable. The variable name can contain only letters and is case sensitive. If the variable name is not allowed (the string contains other characters than letters), `NULL` is returned.

You can also assign a number to the *Multitree*. To assign an integer to *Multitree*, use *inttotree* function:

```
Multitree * fortytwo = inttotree(42);
```

The parameter of this function is of *long long int* type.

To assign floating point to a tree, use *doubletotree*:

```
Multitree * fourpointtwo = doubletotree(4.2);
```

The parameter of this function is of *long double* type.

In a similar way as variables you can also create constants. Difference between variable and constant is that constants have also numerical value. To assign constant, use *treeconstant* function:

```
Multitree * answer = treeconstant("answer",
                                  fortytwo);
```

The first parameter is name of the variable, second is the pointer to *Multitree* with the number that you want to associate with the constant. You can of course use previous two functions in arguments:

```
Multitree * answer = treeconstant("answer",
                             inttotree(42));
```

Suppose we already have several *Multitrees* with numbers and variables and we want to build something more complicated. We can add, subtract, multiply, divide and raise them to power. The functions that do these actions are very similar to each other. Suppose we have `Multitree * a` and `Multitree * b` with some content. We can use any of the following functions:

```
Multitree * result = treeplus(a, b);
                         /* result = a + b */
Multitree * result = treeminus(a, b);
                         /* result = a - b */
Multitree * result = treemul(a, b);
                         /* result = a * b */
Multitree * result = treediv(a, b);
                         /* result = a / b */
Multitree * result = treepow(a, b);
                         /* result = a ^ b */
```

The parameters are pointers to *Multitrees* and the returned pointer to *Multitree* contains the resulting expression in its canonical form[2]. You can combine all these functions together to make more complicated expressions. For example if you want to make `(a+b)^2` you can write:

2 Will be explained later.

```
Multitree * result = treepow(treeplus(a, b),
                                 inttotree(2));
```

In a similar way, it is possible to create expressions containing functions. SymCe supports the following functions: sine, cosine, tangent and (natural) logarithm. Their use is again very easy:

```
Multitree * result = treesin(a); /*sine of a*/
Multitree * result = treecos(a); /*cosine a*/
Multitree * result = treetan(a);
                              /*tangent of a*/
Multitree * result = treeln(a); /*logarithm a*/
```

The parameter is *Multitree* pointer and the result is again pointer to a new *Multitree*.

When we have the expression that we want to work with, we can use one of several functions on it.

We can for example copy it to another place. To create a copy of an expression we use *copy_tree*:

```
copy=copy_tree(original);
```

The parameter is a pointer to *Multitree* we want to copy and returned value is a pointer to a copy of that tree. You can also write `copy = original`, but in that case both *copy* and *original* will point to the same memory location and therefore to the same expression. This of course means that when one expression is changed, the second is changed too. Since almost none of the functions work with the expression in-place, this "copying" is unlikely to have unexpected results, but this cannot be guaranteed and making such copies is not recommended.

Another task that we might want to to do with two expressions is comparing them. We use *treecmp* function:

```
int result = treecmp(a, b);
```

Its parameters are pointers to *Multitrees* and the result is *int* – zero, if they are identical. It is not enough that expressions have same meaning (such as `(a+b)^2` and `a^2+b^2+2*a*b`). They have to be completely identical. If they are not identical, nonzero is returned.

If we want to differentiate our expression, we can do it. We use *treediff* function to do this task. Suppose we want to find the first derivative of `x^2` with respect to `x`. We can write something like:

```
Multitree * x = treevariable("x");
Multitree x2 = treepow(x, inttotree(2));
Multitree * d = treediff(x2, x);
```

The variable `d` now contains an expression that is first derivative of `x^2`. The *treediff* function's first parameter is a pointer to *Multitree* containing expression you want to differentiate. Its second parameter is again pointer to *Multitree* that contains (only) a variable with respect to which the first expression should be differentiated. The result is returned (in this particular case it is `2*x`).

You may also want to replace a variable in an expression with another expression. This can be done using *tree_replace_var* function. The following statement replaces all occurrences of `x` in `x2` with `y`.

```
Multitree d = tree_replace_var(x2, x, y);
```

So if `x` contains only variable `x`, `y` contains variable `y` and `x2` is `x^2` from the previous example, the result (stored in `d`) will contain `y^2`. All parameters of *tree_replace_var* are pointers to Multitree and the second one has to contain variable only.

Sometimes it is useful to expand the multiplied terms in an expression. To expand such multiplication, you can use *treeexpand* function.

```
c = treepow(treeplus(a, b), inttotree(2));
d = treeexpand(c);
```

Now `d` will contain expression `(2*a*b)+(a^2)+(b^2)`. The function *treeexpand* works with powers (`x^a` such as `(a+b)^2` is expanded to `(2*a*b)+(a^2)+(b^2)` – it is not left in its biggest form such as `a*a + a*b + a*b + b*b`) and with multiplications (`(a+b)*(c+d)` is expanded to `(a*c+b*c+a*d+b*d)`. Function *treeexpand* has one parameter (again *Multitree* pointer) and returns its parameter expanded.

The last thing you can do with the expression stored in the *Multitree* is finding its Taylor polynomial. This can be done only with expressions that contain only one variable. To find first *n* terms of a Taylor polynomial you can use *treetaylor* function:

```
result = treetaylor(treecos(x), inttotree(0),
                                          10, x);
```

finds first ten terms of Taylor polynomial with center in zero. First parameter is *Multitree* pointer to an expression to which you want to find the Taylor polynomial, second one is its center and it is again *Multitree* pointer that must contain only number, third parameter is a number of terms that should be calculated and the last parameter is a variable by which the first expression should be differentiated. The result is pointer to *Multitree* containing the desired Taylor polynomial. The resulting Taylor polynomial is in variable "`X`" (that is capital `X` – you can change it using *tree_replace_var*).

If we get the expression we were looking for, we probably want to print it out. It is possible to convert an expression stored in the tree to a string (*char* pointer). To do this use *treetostring* function. It gets pointer to *Multitree* as a parameter and returns the string representation of it. There are certain things that can be little confusing when reading the output string. Remember, that operators always have parenthesis around their operands. Therefore `(sin 2*x)` must be read as `(sin 2)*x`, since `2*x` would have parenthesis around itself. Rule is: If you see an operator, it has parenthesis around itself (However there can be one parenthesis for more than one operator – for example for `(a+b-c+d)`. This is because in internal representation of this expression there is only one operator). Sometimes also numbers have parenthesis around themselves. It is also needed to know, that for constants only its name is outputted, not its value. When *treetostring* gets `NULL` pointer as its argument, it outputs `Error tree.`

When working with expressions, we can assign an expression to a variable that already has an expression. The previous content becomes inaccessible in our program, but the garbage collector will take care of it:

```
a = <some_tree>;
b = <some_other_tree>;
a = copy_tree(b);
```

Here `a` was pointing to some tree and `b` to some other tree. When `b` is copied to `a`, the expression that was in `a` can become lost. This is what garbage collector takes care of.

If any of the functions you use has any problems, `NULL` is returned as the result. All functions can accept `NULL` as any of their pointer parameters (and return also `NULL` in that case), so that if there is a problem in one of the functions “inside”, the outer functions behave deterministically and should not fail. However if construction of an expression fails (basically because of low memory), *abort* is called and the whole program terminates – there is not much of what can be done in that case.

2.2.2 Overview of functions:

Function	Description
Multitree * inttotree(long long int i)	returns an expression created from an integer
Multitree * doubletotree(long double d)	returns an expression created from floating point number
Multitree * treevariable(char * name)	returns an expression (containing variable) created from a string with variable name
Multitree * treeconstant(char * name, Multitree * number)	returns an expression (containing constant) created from a *name* and *number*
char * treetostring(Multitree * tree)	converts an expression to string (*char* pointer)
Multitree * treeplus(Multitree * first, Multitree * second)	adds two expressions (returns result)
Multitree * treeminus(Multitree * first, Multitree * second)	subtracts two expressions (returns result)

Function	Description
Multitree * treemul(Multitree * first, Multitree * second)	multiplies two expressions (returns result)
Multitree * treediv(Multitree * first, Multitree * second)	divides two expressions (returns result)
Multitree * treepow(Multitree * first, Multitree * second)	raises the *first* expression to *second* (returns result)
Multitree * treesin(Multitree * argument)	returns sine of the *argument*
Multitree * treecos(Multitree * argument)	returns cosine of the *argument*
Multitree * treetan(Multitree * argument)	returns tangent of the *argument*
Multitree * treeln(Multitree * argument)	returns natural logarithm of the *argument*
Multitree * copy_tree(Multitree * tree)	returns a full copy of the *tree*
int treecmp(const Multitree * first, const Multitree * second)	compares *first* expression with *second*, returns zero if they are identical, nonzero otherwise
Multitree * treediff(Multitree * expression, Multitree * variable)	differentiates *expression* with respect to *variable*, returns result
Multitree * tree_replace_var(Multitree * expression, Multitree * variable, Multitree * by_what)	replaces *variable* in *expression* by *by_what* expression (returns result)
Multitree * treeexpand(Multitree * tree)	expands *tree* (returns expanded tree)
Multitree * treetaylor(Multitree * tree, Multitree * center, unsigned short int terms, Multitree * variable)	returns expression containing *terms* terms of Taylor polynomial (in “X”) of *tree* with center in *center*. *tree* is differentiated with respect to *variable*.

2.2.3 Canonical form

All expressions that are built using Multitree functions are “simplified” before they are returned. Since no one really knows, what simplified really means, they are just transformed to something called canonical form. When transforming to a canonical form, several operations are performed with the expression.

- If expression contains sine, cosine or tangent of zero, it is replaced with corresponding value. Same holds for logarithm of one.

- Trivial powers are replaced by their values. Expressions `1^x`, `x^0`, `0^x` and `x^1` are replaced with `1`, `1`, `0` and `x` respectively, where `x` is any expression. They are replaced in this order, so `0^0` is replaced (as `x^0`) by `1`.
- Terms that differ only in multiplicative constant are added together: if `a` and `b` are numbers and `x` is an expression, `a*x + b*x` is replaced with `(a+b)*x`.
- Similar holds for multiplication – when two operands have the same base: `x^a*x*b` is replaced with `x^(a+b)` if a and b are numbers.
- If anything is multiplied by zero, the whole multiplication is replaced by zero.
- Zero is dropped in sums. Same holds for one in products (`a+0` and `a*1` are replaced with `a` only)

The last three transformations are more important to internals of SymCe but are presented here for completeness' sake.

- Unnecessary parentheses are removed around additions and multiplications. When several terms are added or subtracted and some parts are in parentheses (such as `(b-c)` or `(e+f)` in `a+(b-c)-(e+f)`), those parentheses are removed (and signs are switched when needed, so our example becomes `a+b-c-e-f`). This is done to bring all terms to one level.
- Same holds for multiplication, only `+` is replaced by `*` and `-` by `/`. Expression `a*(b/c)/(e*f)` changes to `a*b/c/e/f`. Both transformations work also with nested parenthesis.
- Operands of addition and multiplication are sorted. The order is arbitrary and it is only to have expression in well defined state. It is useful to have numbers first, so for example `x*2` is sorted to `2*x`.

2.3 Numbers in SymCe

All functions that work with expressions in SymCe are using a type of numbers called *Number* and functions designed to interface with it. This allows user to work with any implementation of numbers he or she chooses if an implementation of several basic functions working with numbers is provided.

Functions that work with expressions in SymCe work with pointers to a type called *Number*. This can be any type of number user wants. All pointers to *Number*s are initialized before they are used in SymCe by function *construct_number*. This function prepares space for number and may initialize any internal structures that used implementation of *Number* needs. The *Number* is then usually filled with some number either created from some built in type (*int*, *double*) or taken as a result of some number operation.

Functions that work with numbers always get a pointer to *Number* and return the result in an output parameter. Output parameter should always be an initialized number (and in SymCe functions it always is). This is of course true also for input parameters. Most of these functions do not return any value.

After the number is initialized, it should be filled with some value. To do that one of the following functions is used:

```
void inttonum(Number * dest, const int src)
void linttonum(Number * dest,
                        const long long int src)
void doubletonum(Number * dest,
                        const double src)
void ldoubletonum(Number * dest,
                        const long double src)
```

All of these functions take its second parameter and fill the number that is in the first parameter with its value. In SymCe all of *dest* pointers point to the memory initialized by *construct_number* function.

Another way to assign to a number its value is using *numbercpy* function. It works very similarly to *strcpy*. It copies its second parameter to the first.

```
void numbercpy(Number * dest,
                        const Number * src)
```

Sometimes it is needed to create an opposite number to one we have or find an inverse. The functions

```
void numberopp(Number * dest,
                        const Number * src)
void numberinv(Number * dest,
                        const Number * src)
```

do just this. The first one finds an opposite number to the one that is given in *src* parameter and places the result to the *dest*. The second function does the same, but inverts the number instead of creating opposite.

Both tasks can be of course done using subtraction from zero or by dividing one with the number. It is also possible that the implementation of these functions will do just that. However it seems to be nicer to have separate functions for this.

Of course there also have to be operations with numbers. The functions that perform some of the basic operations with numbers are:

```
void numberplus(Number * c, const Number * a,
                              const Number * b)
void numberminus(Number * c, const Number * a,
                              const Number * b)
void numbermul(Number * c, const Number * a,
                              const Number * b)
void numberdiv(Number * c, const Number * a,
                              const Number * b)
void numberpow(Number * c, const Number * a,
                              const Number * b)
```

These functions replace usual mathematical operators (*pow* is for raising to power that is often – but not in C – written as ^). They work as

```
c = a <operator> b
```

When user wants to use his or her own implementation of numbers, it is crucial to write these functions in such a way so that one variable could be stored in more than one parameter (such as `numberplus(a, a, b)`). Otherwise these functions can have unpredictable results.

Numbers can also be arguments to mathematical functions. Therefore SymCe needs some method of computing value when numbers are used in such a way. There are functions to do just that:

```
void numbersin(Number * dest,
                         const Number * src)
void numbercos(Number * dest,
                         const Number * src)
void numbertan(Number * dest,
                         const Number * src)
void numberln(Number * dest,
                         const Number * src)
```

These functions calculate the requested function value with *src* as parameter and store the result in *dest*.

Numbers can also be compared. To compare two numbers user can use *numbercmp* function. It takes two numbers (pointers to *Number*) as parameters and returns integer that is zero if they are equal, less than zero, if first parameter is smaller, and greater than zero if first parameter is larger than the second one. This is similar to the *strcmp* function from standard C library.

Number also has to be printed out sometimes, so there is also function named *numtostr* that converts a number to a string. It takes number pointer as a parameter and outputs a *char* pointer to a text representation of that number. It should use *GC* to allocate memory for the string.

Another function that might be used to output numbers is *numtodouble*. This one converts number to *long double*, if it is possible. If it is not, the result is undefined.

The last function that is used with numbers is *number_is_int*. This function returns one if the number in its parameter is an integer and zero if it is not.

These last two functions are used in expanding of the expression when expanding something raised to an integer (such as (a+b)^4). Function *number_is_int* is used to find out if the exponent is an integer and *numtodouble* is used to convert *Number* so that it is easier to work with it.

Functions that work with numbers and that user should provide if it is desired to use some other (better) implementation of numbers:

Function	Description
void numbercpy(Number * dest, const Number * src)	copies number from *src* to *dest*
void inttonum(Number * dest, const int src)	stores value of *src* to *dest*
void linttonum(Number * dest, const long long int src)	stores value of *src* to *dest*
void doubletonum(Number * dest, const double src)	stores value of *src* to *dest*
void ldoubletonum(Number * dest, const long double src)	stores value of *src* to *dest*
int numbercmp(const Number * first, const Number * second)	compares numbers *first* and *second* returns zero for equal, negative value for *first*<*second* and positive value for *second*>*first*
void numberplus(Number * dest, const Number * first, const Number * second)	stores (*first* + *second*) to *dest*
void numberminus(Number * dest, const Number * first, const Number * second)	stores (*first* - *second*) to *dest*
void numbermul(Number * dest, const Number * first, const Number * second)	stores (*first* * *second*) to *dest*
void numberdiv(Number * dest, const Number * first, const Number * second)	stores (*first* / *second*) to *dest*

Function	Description
void numberpow(Number * dest, const Number * first, const Number * second)	stores (*first* ^ *second*) to *dest*
void numberopp(Number * dest, const Number * src)	stores opposite of *src* to *dest*
void numberinv(Number * dest, const Number * src)	stores inverted *src* to *dest*
void numbersin(Number * dest, const Number * src)	finds sine of *src* and stores it to *dest*
void numbercos(Number * dest, const Number * src)	finds cosine of *src* and stores it to *dest*
void numbertan(Number * dest, const Number * src)	finds tangent of *src* and stores it to *dest*
void numberln(Number * dest, const Number * src)	finds natural logarithm of *src* and stores it to *dest*
int number_is_int(const Number * number)	returns one if *number* is an integer, zero otherwise
long double numtodouble(const Number * number)	returns value of *number* as *long double*
Number * construct_number()	initializes number – allocates memory and prepares eventual internal structures
char * numtostr(Number * number)	returns string that contains value of number

2.4 Inside SymCe

2.4.1 Multitree

All SymCe expressions are stored in trees. This kind of tree is called *Multitree* in SymCe, because it is not only binary tree but one node can have any number of subnodes. *Multitree* is a structure with two members. The *symbol* (of type *Symbol*) and pointer to *ListOfChilds* structure called *childs* (they are deliberately not called children). *Symbol*s are data in the tree. *Symbol* can hold variable, constant, number, operator or function. See the description of *Symbol* structure below. *ListOfChilds* is a pointer to linked list of the childs of the current node. Childs of the node represent arguments of functions or operands of operators. Number of childs of any specific node depends on the type of that node (that is on content of *symbol* variable). If the symbol is number, constant or variable, the node has no childs (this kind of node is called leaf). If it is function, the node has one (SymCe does not know about functions with more than one arguments) and if it is an operator the node has at least two childs.

ListOfChilds is a plain and simple linked list. It contains "traditional" *next* pointer and it contains two data items. The first one is *subtree* (pointer to *Multitree* representing the expression that is "below" the current tree) and *int* called *inverted* (used as boolean) that modifies the meaning of the parent node.

All that SymCe does is basically just changing tree of one expression to another tree of another expression.

2.4.2 Symbol

Symbol is a structure that holds actual data in the tree. It is in all nodes of the tree. It can contain variable, operator, function, number or constant. One can think of symbol as of variant record.

The *type* member inside *Symbol* indicates what is really stored in the *Symbol*. It can be one of the following: `NUMBER`, `FUNCTION`, `OPERATOR`, `VARIABLE` or `CONSTANT`. Depending on the symbol type, one of the following members of *Symbol* is used (although all of them are always present).

func – function stored in the symbol. It can be one of `LN`, `SIN`, `COS`, `TAN`.

oper – operator stored in the symbol. It can be one of `PLUS`, `MULTIPLY`, `POWER`. There is no minus or division operator in symbols. The reason for this is discussed below (and it has to do with an *inverted* item in *ListOfChilds*).

All of the previous members of *Symbol* structure are of enumerated type. You can store only one of the predefined values in them. *Symbol* has also two pointers that are used when storing numbers, variables or constants.

number – pointer to *Number*. It, obviously, serves to store numbers.

name – *char* pointer that stores a name of a variable or a constant

To see what a symbol really stores, first look at *type* and then look at the corresponding member of the *Symbol* as set in the following table:

What is stored in *type* variable	What other variable to look at
NUMBER	Look at contents of memory pointed to by *number* pointer to see what number is stored in the *Symbol* (to interpret the content of this memory refer to documentation of used implementation of numbers). Other parts of symbol are undefined.
FUNCTION	Look at value of *func* variable. It will tell what kind of function is in the *Symbol*. Other parts of *Symbol* are undefined.
OPERATOR	Look at value of *oper* variable. It will tell what kind of operator is in the *Symbol* (similar to function) Other parts of *Symbol* are undefined.
VARIABLE	The *name* stores a `NULL` terminated string (consisting only of letters) containing a name of the variable. Other parts of *Symbol* are undefined.
CONSTANT	This is combination of variable and number. The name of the constant is in memory pointed by *name* and its value is stored in memory pointed to by *number*. Most of the time, when working with constants, the value is not really important. It is however needed when trying to calculate value of an expression.

2.4.3 How is expression stored in tree.

If the expression is a number, variable or a constant, *Multitree* that stores it has *symbol* filled appropriately and its *childs* pointer is ignored (and is set to `NULL`).

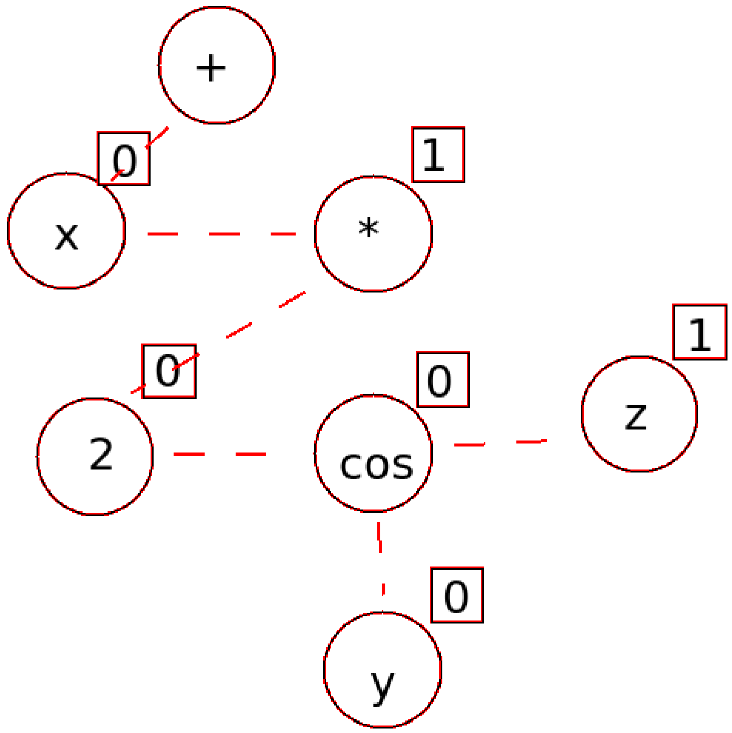
If the expression is a function with an argument (argument is again an expression), the *symbol* in the *Multitree* is set to appropriate function and there is exactly one item in the *childs* linked list. The child has *inverted* set to zero, *next* is set to `NULL` (there is only one item in the list) and its *subtree* pointer is pointing to a *Multitree* representing the argument of the function.

When the expression is an operator with operands, things become more interesting. The first case is power operand (*oper* in the tree's symbol is set to `POWER`). Its *childs* variable is a linked list with two items. Both have their *inverted* members set to zero and the first has its *subtree* pointed to a *Multitree* that contains base and the second to the one containing exponent.

If the operator is `PLUS` or `MULTIPLY`, the *childs* linked list can have as many items as needed. These operators also represent subtraction and division respectively. So that expressions such as `a+b-c` or `a*b/c` have all three `a`, `b`, `c` operands as childs of one node. This is where *inverted* comes to play – those operands that are subtracted (or it is divided by them) have *inverted* set to one.

To see an example of more complex expression stored in the tree, look at the following picture.

We want to store `x - 2*cos(y)/z` in the tree.



As you can see, every part of the expression has its own node (represented by circle). First we have “plus” node. It has `x` as its first child (that is its `childs->subtree` points to a *Multitree* containing `x`) and `*` as a second child (that is its `childs->next->subtree` points to the *Multitree* containing multiplication). The numbers in the squares represent *inverted* flag in *childs*. It is zero for `x` (`childs->inverted` is zero), because `x` is added and it is one for `*`, because it is subtracted (`childs->next->inverted` is one). It is similar for node with multiplication. Since there is division by `z`, its *inverted* flag is set to one. The horizontal lines mean, that childs of one node are in a linked list although logically they are all childs of the node above (that is there could be lines from `*` to `cos` and `z`).

2.4.4 Numbers

The default representation of numbers is rather trivial one. Numbers are of *Number* type. *Number* is a structure, that has three members: *type*, *fraction* (type *Fraction*) and *real* (type *Real*). The *type* member indicates whether the value is stored in *fraction* or in *real*. *Real* is simple *long double*. *Fraction* is another structure that contains two integers (*p* and *q*, both *unsigned long long int*) and a *sign* (*enum* that can be `POSITIVE` or `NEGATIVE`). *p* and *q* are numerator and denominator of a fraction respectively, *sign* is its sign. For zero, the *sign* is set to `NEGATIVE`[3]. Fractions are kept in simplest terms. The result of operation with *Number*s is usually again *Fraction*, when both operands are *Fraction*s (raising to power is an exception to this) and *Real* when at least one of the operand is *real*.

Functions that work with numbers are discussed in previous sections of this text. If user wants to use his or her own implementation of the numbers, he or she needs to provide implementation of a functions in `Number.h`. For current implementation refer to the source.

There are also functions that work with fractions. First idea was to write fractions as independent as numbers are so that fractions could have been replaced in a same way as numbers; however, some of number functions (namely *numberpow*, *numtostr* and *numtodouble*) work with internals of fraction.

2.4.5 Symbol functions

Most of the tree functions work with data inside *symbol*s (there is a switch that depends on symbol type of the particular tree); however, there are two functions that are provided to use with *symbol*s if there is need (and they are of course used in some SymCe functions. The first one is

```
int symbolcmp(const Symbol * first,
                             const Symbol * second)
```

that compares two symbols. It looks at symbols in its parameters and returns number smaller than zero if the first parameter is smaller than second, zero if they are equal and greater than zero if the first one is greater than the second. It is same as in standard *strcmp* function (and in some other *cmp* functions in SymCe).

The order of the symbols is quite arbitrary with a small exception. If one of the symbols is a number and the other is not, the one with number is smaller. This is useful for reducing tree to canonical form, where sometimes it is needed to take coefficient (that is – a number) from some part of the tree and it is taken as the first item in the list, since it is sorted there as the smallest.

The second function is

```
void symbolcpy(Symbol * dest,
                             const Symbol * src)
```

that simply copies content of symbol from *src* to *dest*.

2.4.6 Functions that manipulate with Multitrees

Almost all functions that manipulate with *Multitree*s get the pointer to a *Multitree* as a parameter and return a pointer to a newly created one. There are of course some exceptions and those will be mentioned below.

3 This is an arbitrary decision because of somewhat dubious historical reasons.

```
Multitree * copy_tree(Multitree * tree)
```

This function takes pointer to *Multitree* and creates an exact copy of that tree.

```
void delete_tree(Multitree * tree)
```

This function says to garbage collector that it is OK to release the memory occupied by the tree. It first works on all subnodes of the current node and then marks the current node for deletion (using `GC_FREE` macro of the garbage collector). This function can be completely removed (and is present only for historical reasons). The principle "first work on subnodes then on current node" that can be seen in this function is common to many tree handling functions in SymCe.

```
int treecmp(Multitree * first,
                         Multitree * second)
```

This functions compares two canonicalized trees (trees in normal form). Its return value is similar to *strcmp* – zero for equal trees, negative if *first* is smaller and positive if *second* is smaller. It firstly compares the symbols in trees (using *symbolcmp* function) and if they are identical then it goes through childs (first different child determines the result – `NULL` is smaller than an existing child). This is used when sorting operands of commutative operators and "numbers first" principle from *symbolcmp* is used here. This function can also be run with trees that are not in their canonical form as its parameters but this have no practical use.

```
char * treetostring(Multitree * tree)
```

This function creates a string representation of a tree that can be printed out. For leaf it outputs its value, for function it outputs its name and then calls itself on the argument (and concatenates the strings) and for operator it outputs opening parenthesis, goes through operands and puts a correct operator sign between then and then closes the parenthesis. Memory for the string is allocated as one hundred of *char*s and if it is not enough, it gets doubled. The current allocated amount is stored in *allocated* variable and how much of it is used is stored in *now_used* variable.

```
void rehang_tree(Multitree * tree)
```

This is one of few functions that work with tree in place (do not copy it). It finds all plus (multiply) nodes with plus (multiply) subnodes and merges them to one. This function first finds what kind of tree it has (using switch on `tree->symbol.type` – this is common to many SymCe functions), then calls itself to sub nodes (if any) and then does its work on the current node (if the node is of type `+` or `*`). It first separates subnodes to two groups – those with same symbol (called *compatible*) and others (called *incompatible*). Then it goes through *compatible* group (which has all subtrees with same symbol as current tree). If element in that group is marked as inverted, all subnodes of that element have their *inverted* flag switched. Then subnodes of elements in this group are connected to *incompatible* and all, together, are connected to current tree as subnodes.

Example: in expression `(a+b)-(c-d)+e+f` the incompatible group has two members: `e` and `f`. Compatible has `(a+b)` and `(c-d)`. `(c-d)` is inverted (it had `-` in front of it) to `(d-c)`. Then `a`, `b`, `c`, `d`, `e` and `f` are merged to one list and the result is `a+b-c+d+e+f`.

```
void sort_tree(Multitree * tree)
```

This function sorts the childs of `+` and `*` nodes. It uses *cmp_tree* as comparison function and in case the subtrees are identical, it looks at *inverted* flags. Used sorting algorithm is *minsort* (there are usually not many subnodes to be sorted, so speed is not a problem and also they are most often already sorted). This is another function that does its work without copying the tree (since it would be too costly and it is not necessary).

```
Multitree * reduce_tree(Multitree * tree)
```

This is one of more complex functions in SymCe. It was separated to more functions. If it gets leaf it just copies it. If it gets function, it processes its argument first. If reduced argument is not number it connects it as argument to copy of current node and returns it. Otherwise it calls *function_values* that either return tree with value of the function or same tree as in previous case. This result is then returned.

If the tree is an operator then it does quite much work. For operator `^` *reduce_tree* just checks some trivial cases (see trivial powers on page 18) and returns the appropriate result (zero to zero is one here – this can of course be, quite trivially, changed – for example to returning `NULL` as an undefined tree). In nontrivial case only childs are reduced and power is returned.

In case of operators + or * it goes through *childs* and adds (or multiplies) what can be added (or multiplied). Example: `x+2*x+3*x-4*x` is reduced to `2*x`, `x*x^2*x^3/x^4` is reduced to `x^2`. This is done in *add_and_multiply* function (that was created to keep *reduce_tree* in manageable size, otherwise its content could be put directly to *reduce_tree*).

Function *add_and_multiply* first separates the numbers from the list. Then all remaining childs are torn to two parts: *coefficient* (that is number) and *rest* (tree). These two parts are then put to a linked list of something called *Tuplets* in SymCe (tearing and putting to *tuplets* are done by *childs_to_tuplets* function). In our examples (x+2*x+3*x-4*x and x*x^2*x^3/x^4), the coefficients would be 1, 2, 3 and -4 and trees will be always x. For -x the coefficient would be -1. Separation (for plus trees) depends on the tree being sorted, since coefficients should be in its places (the numbers should be the first part in the tree – `x*y*4` is sorted to `4*x*y`, so that number is first and then the tree can be easily separated to `4` and `x*y`).

Then this list of *Tuplets* is processed – coefficients are added for identical trees (*treecmp* is again used) using *add_tuplets* function (basically, if there are two or more tuplets with same tree, its coefficients are added and they are replaced by one tree). *remove_null_tuplets* removes those tuplets that have coefficient set to zero. Resulting tuplets are transformed back to list of childs, joined with numbers (those separated from the *childs* at the beginning) that were added or multiplied as needed. If the coefficient of a tuplet is `1` or `-1` then this coefficient is not put back to tree (so that result that would be `1*x` is only `x`). This list of childs is then attached back to operator if there are at least two remaining childs.

If there are zero remaining childs, zero is returned for addition and one for multiplication. If there is one child, its subtree is returned if it is not inverted. If it is inverted, then its subtree is multiplied by `-1` for addition and raised to `-1` for multiplication. Example: `a-b-a` would return *plus* with only one child (`b` inverted). That would be incorrect tree, so instead `-1*b` is returned.

```
Multitree * canonicalize_tree(Multitree * tree)
```

This functions calls previous three functions in a cycle until the expression does not change. It does so because after rehanging there might be more parts that can be added together using *reduce_tree* which can again produce some tree that can be rehanged and so on.

Example: `x*(0+x*y)` is reduced to `x*(x*y)` that is rehung to `x*x*y` and that is reduced to `x^2*y`.

```
Multitree * diff_tree(Multitree * tree,
                              char * variable)
```

Creates a tree that contains first derivative of the original tree with respect to *variable*. As most of the procedures it goes first through subtrees, creates first derivatives of them and then connects them to appropriate result using copies of both original and differentiated subtrees. If the current node is operator, it calls helper function to do the job.

This function is called from *treediff* function (where *variable* is *Multitree* and is checked to be really variable).

```
Multitree *
replace_variable_tree(Multitree * original,
                char * variable, Multitree * with)
```

This function works in a same way as *copy_tree* with one difference. If current node is a variable with name same as in argument, the tree *with* is copied in its place. This function is called from *tree_replace_var* function that checks its parameters.

```
Multitree * expand_tree(Multitree * tree)
```

expand_tree takes multiplication nodes and expands them (using *expand_multiply*): from `(a+b)*(c+d)` it creates `a*c+a*d+b*c+b*d`. First it separates the operands to four groups: *upper_normal*, *upper_plus*, *lower_normal* and l*ower_plus*. *Upper* are those that are not inverted, *lower* are inverted. *Plus* are those operands of current node that have plus as their topmost node, *normal* are others. In (a+b)*c/d/(e+f), (a+b) is *upper_plus*, c is *upper_normal*, d is *lower_normal* and (e+f) is *lower_plus*. The normals are converted to `(0+normal)` and attached to the corresponding *plus* lists. The function then goes through both *plus* lists and expands them. It takes first two items in the list, multiplies them (each term from first with each term from the second) and puts them back to the list as a new item (but it calls *canonicalize _tree* before putting them back to prevent exponential increase of number of terms). It repeats until there is only one item left. `upper_plus/lower_plus` is then returned.

Example:

- original: `(a+b)*c/d/(e+f)`
- separated:
  - *upper_normal*: `c`
  - *upper_plus*: `(a+b)`
  - *lower_normal*: `d`
  - *lower_plus*: `(e+f)`
- normals attached:
  - *upper_plus*: `(0+c)`, `(a+b)`
  - *lower_plus*: `(0+d)`, `(e+f)`
- expand:
  - *upper_plus*: `0*a+0*b+c*a+c*b` canonicalized to `c*a+c*b`
  - *lower_plus*: `0*e+0*f+d*e+d*f` canonicalized to `d*e+d*f`
- result: `(c*a+c*b)/(d*e+d*f)`

expand_tree also takes all power nodes and if exponent is an integer then it expands them (using *expand_power*). For example from `(a+b)^3` it creates `a^3+b^3+3*a^2*b+3*a*b^2`. This is done by making powers of powers of two of original tree and then multiplying appropriate powers together. Example: `(a+b)^9` is done by computing `(a+b)^1`, `(a+b)^2`, `(a+b)^4` and `(a+b)^8` (each of course already expanded by multiplying previous term with itself) and then selecting `(a+b)^1` and `(a+b)^8` and multiplying them (using again *expand_multiply*).

```
Number * numeric(Multitree * tree)
```

This function finds numeric value of the tree if it does not contain any variable. If it finds a variable in the tree, it returns `NULL`.

```
Multitree * give_me_tree_with_two_childs()
```

Creates tree representing `0+0`. Often a tree with several subnodes needs to be created and that means that quite many things have to be done. This function does them (create tree, create two childs, and link them together and to the tree, create subtrees and put `NULL` to correct places) all in one place. However, this function is provided only for historical reasons – new function called *give_me_tree_with_two_incomplete_childs* was created for convenience. It creates incomplete structure with one tree node and two childs whose subtrees are `NULL`. This is not a valid tree, but the structure returned from this function is immediately completed in function that called it. There are also two other *give_me* functions (*give_me_empty_child* and *give_me_empty_tree*). They allocate memory for appropriate structure and return pointer to it. These functions are trivial.

```
Multitree * taylorize(Multitree * tree,
                      Multitree * center,
                      unsigned short int terms,
                      char * variable)
```

Creates first *terms* terms of Taylor polynomial It computes derivatives of *tree* in a cycle, replaces *variable* in them with *center* number, finds their numerical values and attach these values with appropriate (nth) factorial and `(X-center)^n` to terms that are attached to plus node. If *numeric* returns `NULL` (because there was another variable in tree than the *variable*), `NULL` is returned. The resulting polynomial is in (capital) `X` that can be replaced using *replace_variable_tree* if needed. This function is called from *treetaylor* that checks some of its parameters so that *taylorize* is not called with wrong parameters.

Functions that build the expression were already described in previous sections of this text. They all work alike. They create new node, attach (copy) eventual childs and run *canonicalize_tree* on the built tree.

Other functions are either previous functions split to more than one (to keep thing manageable (such as *add_and_multiply* – its content could be put to *reduce_tree* but then *reduce_tree* would be too large) or are only helpers (such as functions that work with *Tuplets*) or are trivial (such as *delete_list* or *give_me* functions).

2.5 Compiling SymCe

To build SymCe library and sample SymCe program, run *make* in the directory with source. It will create a shared library called *libsymce.so*, binary called *symce* and a binary called *symce-static* that is statically linked with SymCe library. Source for programs *symce* and *symce-static* is stored in `main.c`. To run *symce*, set `LD_LIBRARY_PATH` to directory containing *libsymce.so*, since *symce* needs functions from there.

To build SymCe you need Boehm's Garbage collector that can be found in *libgc-dev* package on Debian based systems. On other systems, there is usually appropriate package available (*boehm-gc* on Gentoo, *gc* and *gc-devel* on OpenSuse...). Otherwise its source can be found online on http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

Compilation of SymCe is just simple compilation of every `.c` file and then linking all resulting `.o` files together with `-lgc` and `-lm` switch (`Makefile` also includes debugging information in the result).

There is also possibility to use your own implementation of numbers. If you decide to do so, you need to provide functions that are declared in `Numbers.h` (and comment out the definition of *Number* type). Then you compile all files except `Fraction.c`, `Integers.c` and `Numbers.c` and link them with *gc* and with your implementation of numbers.

SymCe was tested only on Linux, but should compile on any system where its requirements (Boehm's Garbage collector and possibly *Numbers*) are provided.

2.6 Sample SymCe program

There is a simple SymCe program that shows some of the SymCe properties. It is stored in `main.c` and is quite simple to understand. Refer to `main.c` for the code.

First it creates several variables (`a` – `f` and `x`). Then it prepares an expression containing `cos(x)` and called `x2`.

Then a third derivative of `x2` is outputted. This shows how you can feed output from one function to other (here result of *treediff* function is used as an input for another *treediff*). Also note, that neither `x` nor `x2` are changed and `x` can be used several times in single statement without any undesired consequences or side effects.

Output: `sin x`

Then there is another example of differentiation and stacking tree functions together. This one sums sine of `x` and tangent of `x` to form `(sin x + tan x)` and finds its derivative.

Output: `(cos x+((cos x^2)^(-1)))`

The next statement shows expansion of multiplication with terms that are multiplied and terms by which others are divided. There are parts containing plus or minus and those that do not. Here is also an example of larger expression stored in SymCe. It is of course not as easy to write an expression in SymCe as it is would be using operators but since there is no operator overloading in C, this is the best way to do it (of course, you can do it in multiple steps with temporary variables to make it cleaner).

Output: `(((a*c)-(a*d)+(b*c)-(b*d))/((e*x)-(f*x)))`

Next part is created to show that expressions are reduced to "normal" form automatically. The expression `0*a-0*b+c` is created and sent to output, but is canonicalized before it is passed to *treetostring* function.

Output: `c`

Next statement is again demonstration of expansion. This time it is `(a+b)^(-3)`.

Output: `(((3*(a^2)*b)+(3*(b^2)*a)+(a^3)+(b^3))^(-1))`

Now there are again four simple examples of how SymCe keeps expressions in normal form. `x-x` is reduced to `0`, `x+x` to `2*x`, `0*x` to `0`. The last one is little more complex (and again you can see longer expression written in SymCe), but is reduced simply to `0`.

Output: `0` , `(2*x)` , `0` and `0` respectively.

Simple differentiation of logarithm is only for showing, that there is also logarithm in SymCe.

Output: `(x^(-1))`

Then there are two Taylor polynomials of `x2` (that has `cos(x)` in it). The difference is in *center*. The first one is centered in `0`, the second one in `1`. Here you can see, that if cosine (and sine) gets zero as parameter, the value is exact and the output contains fractions (value of derivative divided by corresponding factorial) and not floating points. The terms in resulting polynomial are sorted according to *treecmp* and not by their exponent. However, numbers first principle holds also here.

Output:

```
(1+((1/40320)*(X^8))-((1/720)*(X^6))+
((1/24)*(X^4))-((1/2)*(X^2)))

(0.540302-(2.31887e-06*(((-1)+X)^9))
+(1.34004e-05*(((-1)+X)^8))
+(0.000166959*(((-1)+X)^7))-
(0.00075042*(((-1)+X)^6))-
(0.00701226*(((-1)+X)^5))
+(0.0225126*(((-1)+X)^4))
+(0.140245*(((-1)+X)^3))-
(0.270151*(((-1)+X)^2))-(0.841471*((-1)+X)))
```

Next lines of `main.c` contain a commented loop that just finds derivatives of `cos x` over and over again. When it is running, the amount of the memory of the process can be watched and it really does not rise astronomically, so we can see, that garbage collector does its work. No output is done, since this is commented out. Otherwise repeating sines and cosines of `x` (sometimes negative).

Following are two simple tasks (differentiation of `x^2` and expansion of `(a+b)^4`.

Output: `(2*x)` and `((4*(a^3)*b)+(4*(b^3)*a)+(6*(a^2)*(b^2))+(a^4)+(b^4))`

The last one is an illegal division: `1/0`. The returned tree is `NULL`.

Output: `Error tree.`

2.7 What could have been done better

After writing a piece of software, things that could have (and should have) been done differently are found. SymCe is no exception. There are many things that could be better, that should not be there at all, and those that should be there, but are not.

The most obvious thing are numbers. SymCe is very simple project to show that it is even possible to do CAS in C so that the easiest (not completely trivial – that is not some of the built in types) implementation of numbers was chosen. Fractions were already implemented as a small programming exercise, so there was no reason not to reuse them. However, it was soon obvious that having only fractions in the program is not so good idea, so *Number* structure was created to store either fraction or real number (which was plain and simple *long double*).

Another thing connected with numbers is *numeric_function* function. It contains several special cases and if none of them is true, then the value is calculated using *number*something function. This is because the way numbers are implemented is known and because number functions do not test special cases (and not even impossible ones). But the special cases could be dealt with inside number functions (so that *numbersin* returns exact zero when called with zero). But then again, *numeric* can be (quite easily) rewritten to see if there is not something like `sin(2*pi)` in the expression (provided we include `pi` constant) and return exact value also in this case. *Number* functions in this case could only try to see, if the argument is "close enough" to some predefined table values. But the same special cases should be dealt with in *reduce_tree* and its *function_values* function, so that we can "simplify" the tree better. I do not see any elegant way to write both *function_values* and *numeric_function* – they both should know the same special cases but return different results.

Creating expression using provided functions is quite straightforward, but it is more usual to do it using infix notation and operators. It would be convenient to have a function that creates a *Multitree* from a string. This function would have to parse the string and create a whole tree. It would also need a function to create a number from a string. However, no such functions are provided. They are probably the first thing to do in future versions of SymCe.

Next thing that could have been done better is employing more code reuse. Some attempts can be seen in *give_me* type of functions (in older versions every function that needed some structure not only allocated memory alone, but had to build complete structure itself) or in *treefromnum* function (creates a tree from *Number*) – there were many places, where a tree created from number was needed, so there were several lines repeating through the code (not anymore). But there are still parts, where quite large blocks of code repeat. Special example is *expand_multiply* function that for lower and upper parts does exactly same thing. This was done using copy&paste to finish the function quickly and there was a plan to somehow do it in a clean way, but since it worked without any problems, the motto "If it's not broken, do not fix it." caused it to stay the way it is now.

There is also very small number of mathematical functions that SymCe now knows about (only four). New functions can be added (with not too big touches to the program – differentiation being the biggest change), but to add a function one must change code in several places and add even some “user functions”. When creating SymCe, these four functions seemed as enough and for the purposes of showing, that CAS can be done in C, they still are.

3 Conclusion

SymCe is now working piece of software that is (at least should be) simple enough for anyone to use, extend, study and enhance. It shows, that C language can be used for symbolic manipulation (directly in the C language) in quite an acceptable way. Since it does not have operator overloading, the expressions are built using functions, what can be not too straightforward (one can get lost in it), but it is natural enough so that anyone who thinks about using SymCe should be able to do that.

The SymCe was kept as simple as possible (sometimes even with the cost of little functionality), but simplicity that was the main aim of the whole work has been achieved.

4 Bibliography

1. SymPy tutorial. http://docs.sympy.org/tutorial.html (accessed on 20. 7. 2008)
2. GiNaC tutorial. http://www.ginac.de/tutorial/
3. strcmp, manual page
4. Hans J. Boehm: A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (accessed on 20. 7. 2008)
5. Wikipedia contributors: Garbage collection. http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29 (accessed on 20. 7. 2008)
6. Wikipedia contributors: Computer algebra system. http://en.wikipedia.org/wiki/Computer_algebra_system (accessed on 20. 7. 2008)
7. Joel Spolsky: The Law of Leaky Abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html (accessed on 20. 7. 2008)
8. Joel Spolsky: Back to Basics. http://www.joelonsoftware.com/articles/fog0000000319.html (accessed on 20. 7. 2008)
9. the PARI group: PARI/GP. http://pari.math.u-bordeaux.fr/ (accessed on 20. 7. 2008)
10. Christian Bauer, Alexander Frink And Richard Kreckel: Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. Journal of Symbolic Computation (2002) 33, 1–12. http://www.ginac.de/csSC-0004015.pdf (accessed on 4. 8. 2008)
11. Fanwell Kenala Bokosi: Determinants and Characteristics of Household Demand for Smallholder Credit in Malawi. Abstract, http://web.onetel.com/~fanwellbokosi/docs/Abstract.pdf (accessed on 22. 7. 2008)

5 Appendix

List of SymCe files

- Diff.c
- Diff.h
- Expand.c
- Expand.h
- Expression.c
- Fraction.c
- Fraction.h
- Integers.c
- Integers.h
- main.c
- Makefile
- Multitree.c
- Multitree.h
- Mystrcat.c
- Mystrcat.h
- Number.c
- Number.h
- Numeric.c
- Reduce.c
- Symbol.c
- Symbol.h
- Taylor.c
- Tuplets.c
- Tuplets.h